Tiling Decision Trees Using Edge Probabilities

Ashwin Prasad, PhD CSA
ashwinprasad@iisc.ac.in

February 23, 2022

1 Introduction and Notation

The fundamental problem we are faced with is to compute the prediction of a decision tree $T = (V, E, r)$ where V is the set of nodes, E the set of edges and $r \in V$ is the root. For each node $n \in V$, the following are given.

1. $threshold(n) \in \mathbb{R}$ which gives the threshold value for n.
2. $featureIndex(n) \in \mathbb{N}$ which gives the feature index for n.
3. $left(n) \in V$, the left child of n or $\emptyset$ if n is a leaf. If $left(n) \neq \emptyset$, then $(n, left(n)) \in E$.
4. $right(n) \in V$, the right child of n or $\emptyset$ if n is a leaf. If $right(n) \neq \emptyset$, then $(n, right(n)) \in E$.

We use $L \subset V$ to denote the set of leaves.

To compute the prediction of a tree T on an input row x, the procedure in algorithm 1 is followed.

Algorithm 1 Compute Tree Prdiction

```
procedure PredictTree($T = (V, E, r)$, $x$)
    $n \leftarrow r$
    while $n \notin L$ do
        //Compute the predicate of the current node.
        $pred \leftarrow x[featureIndex(n)] < threshold(n)$
        //If predicate is true, move to right child, else left child.
        if $pred$ then
            $n \leftarrow left(n)$
        else
            $n \leftarrow right(n)$
        end if
    end while
    //Return the threshold value of the leaf as the tree prediction.
    return $threshold(n)$
end procedure
```

In order to speed up the tree walk, Treebeard aggregates decision tree nodes into tiles and uses vector instructions to evaluate tiles. Experiments have shown that, in several trees, some leaves are much more likely to be reached than others. Here, we explore how, given this fact, the tree can be tiled to **minimize average inference latency**. Practically, tiling is important from the perspective of both instruction count and locality.

2 Tree Tiling

A tiling $\mathcal{T}$ of the tree $T = (V, E, r)$ with tile size n_t is a partition $\{T_1, T_2, ..., T_m\}$ of the set V such that

1. $T_1 \cup T_2 ... \cup T_m = V$
2. $T_i \cap T_j = \emptyset$ for all $i, j \in [1, m]$ and $i \neq j$
3. $|T_i| \leq n_t$ for all $i \in [1, m]$
4. $\forall l \in L : l \in T_i \rightarrow v \notin T_i \;\; \forall v \in V \backslash \{l\}$
5. Tiles are **maximal**, i.e. if $|T_i| < n_t$, then there is no $v \in V \backslash \{T_i \cup L\}$ such that $(u, v) \in E$ for some $u \in T_i$.
6. Tiles are **connected**, i.e. for an $u, v \in T_i$, there is a (undirected) path connecting u and v fully contained in T_i.

3 The Optimization Problem

We assume that we are given the probabilities of each leaf node of the decision tree (these can easily be computed using the training data). For every leaf $l \in V$, we are given the probability p_l that the leaf l is reached.

We observe that the latency of one tree walk is proportional to the number of tiles that need to be evaluated to reach the leaf. Therefore, the average inference latency can be modelled as the expected number of tiles that are evaluated to compute one tree prediction. More formally, the problem is to find a tiling $\mathcal{T}$ such that the following objective is minimized.

$$\min_{\mathcal{T}} \sum_{l \in L} p_l.depth_{\mathcal{T}}(l)$$

where the minimization is over all valid tilings $\mathcal{T}$ of the tree T, $depth_{\mathcal{T}}(l)$ is the depth of the leaf l given tiling $\mathcal{T}$.

The constraints on the optimization are the constraints listed in Section 2.

4 Dynamic Programming Formulation

In order to formulate the above optimization problem as a dynamic program, we define the following.

1. For each node $n \in V$, we define the absolute probability p_v as

$$p_v = \begin{cases} p_l & \text{if } l \in L \\ p_{left(v)} + p_{right(v)} & \text{otherwise} \end{cases} \tag{1}$$

2. For any tree T, $\mathcal{C}(T)$ represents the set of all valid tilings of T.

3. For every $v \in V$, we define S_v as the subtree rooted at v.
4. For every $v \in V$, we define L_v as the set of leaves of T_v.
5. For a every tile T_i, we define $root(T_i)$ as the node $v \in T_i$ such that v has no incoming edges from any other node $u \in T_i$.
6. For a tile T_i, $out(T_i) \subseteq E$ is the set of edges (u, v) such that $u \in T_i$ and $v \notin T_i$.

For any node $v \in V$, we define

$$cost(v, \mathcal{T}) = \sum_{l \in L_v} p(l|v).depth_{\mathcal{T}}(l)$$

where $\mathcal{T} \in \mathcal{C}(T_v)$.

Then, the objective function, for the tree T_v, can be rewritten as

$$opt\_cost(v) = \min_{\mathcal{T} \in \mathcal{C}(T_v)} cost(v, \mathcal{T})$$

The objective function can then be rewritten in the following recursive form.

$$opt\_cost(v) = \min_{T_0 \in TileShapes(n_t, v)} 1 + \sum_{(n_1, n_2) \in out(T_0)} p(n_1|v)p(n_2|n1)opt\_cost(n_2)$$

where $TileShapes(n_t, v)$ is the set of all tile shapes of size n_t with root v. A straight forward substitution argument shows why the solution to the subproblems (tiling all sub-trees) needs to to be optimal. The objective is now in a form that can solved using dynamic programming.

5 Greedy Algorithm

Intuitively, it seems like the following greedy algorithm also gives the optimal tiling. The algorithm starts at the root and greedily keeps adding the most probable node to the current tile until the maximum tile size is reached.

Algorithm 2 Greedy Tree Tiling

```
procedure TILETREE($T = (V, E, r)$, $n_t$)
    if $r \in L$ then
        return
    end if
    $Tile \leftarrow \{r\}$
    while $|Tile| < n_t$ do
        $e = (u, v) \in Out(Tile)$ such that $p(v|u)$ is maximum and $v \notin L$
        if $e = \emptyset$ then
            break
        end if
        $Tile = Tile \cup \{v\}$
    end while
    for $(u, v) \in Out(Tile)$ do
        $TileTree(T_v, n_t)$
    end for
    return $threshold(n)$
end procedure
```